A Guide to Using OHD's CHPS Development and Testing Frameworks

March 22, 2011

Over the span of about two years OHD-HSEB ported over a dozen models in NWSRFS to FEWS. A handful of the ports were done by rewriting the model in Java. The majority still use the original C/C++ and Fortran algorithms. In both cases, a set of routines and approaches for interfacing the algorithms to FEWS (a development framework) and then testing the ported model to make sure the model's behavior remained consistent (a testing framework) was needed.

The testing and development frameworks described through the following examples are what we used. The details of each framework are described in the accompanying tar compressed package (**outside_test.tar.bz2)**. The package contains:

1. java archives (jars, containing the Java code we used)
2. Sample implementations of the development and testing framework (demonstrated using Eclipse)
3. Documentation in the form of Javadoc

These resources should reduce the amount of time needed to port other algorithms/models to CHPS. It should allow modelers to focus on their algorithms and not the interfacing code (e.g. Input/Output of data, etc.)

Installing and Setting Up the Package

1. Untar the tar file(outside_test.tar.bz2) provided:

   $tar -xvf outside_test.tar.bz2

   Five directories are generated:

   i) libs (basic.jar sacsma_lagk.jar)

   - basic.jar contains all the testing utilities. For running non-OHD models, only basic.jar is needed;
   - sacsma_lagk.jar contains OHD models (SacSma and LagK) which is only needed to run SacSmaTest and LagKTest shown below.

   ii) outside (./Modules ./src ./test)

   This directory contains the demonstration code for how to use the two JAR files above. It has dummy source code (inside src/) and it has junit testing code (inside test/) to test dummy source code and two real models (SacSma and LagK). Modules/bin/lagk is the lagk model binary executable which was compiled from Fortran and C code.

iii) javasrc

This contains the Java source code for external users to view our code.

iv) javadoc

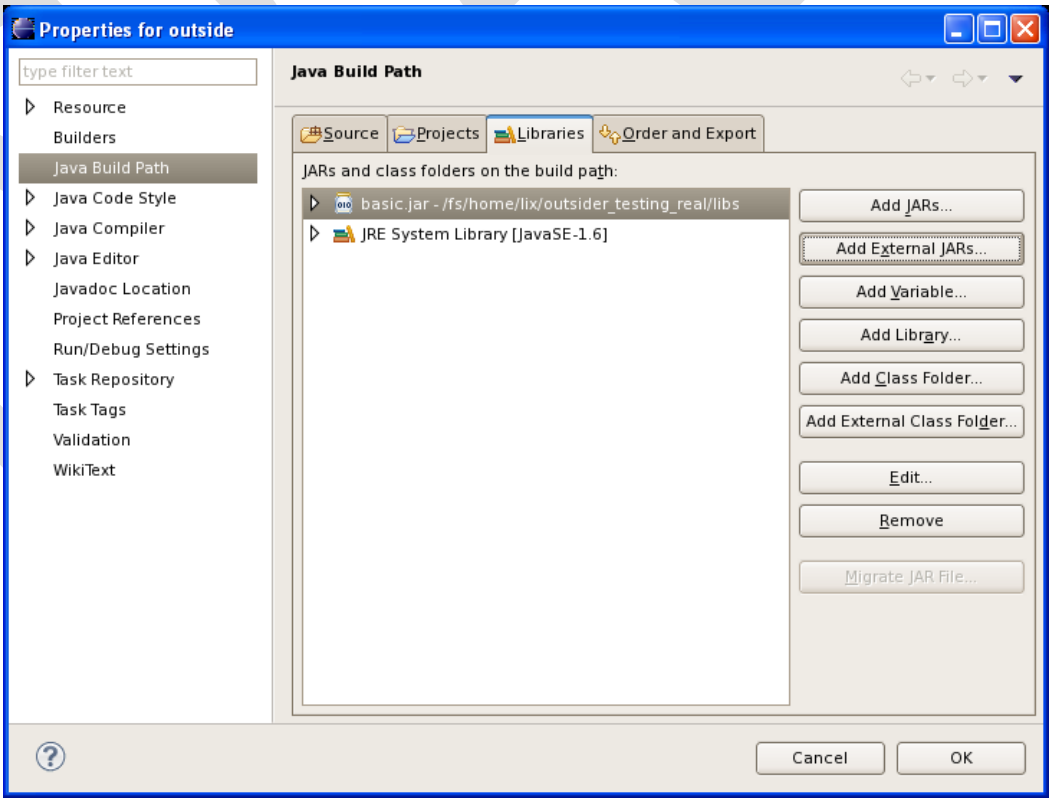
This directory contains JAVA API documents related to classes in basic.jar.

v) wrappedNwsrfsModels

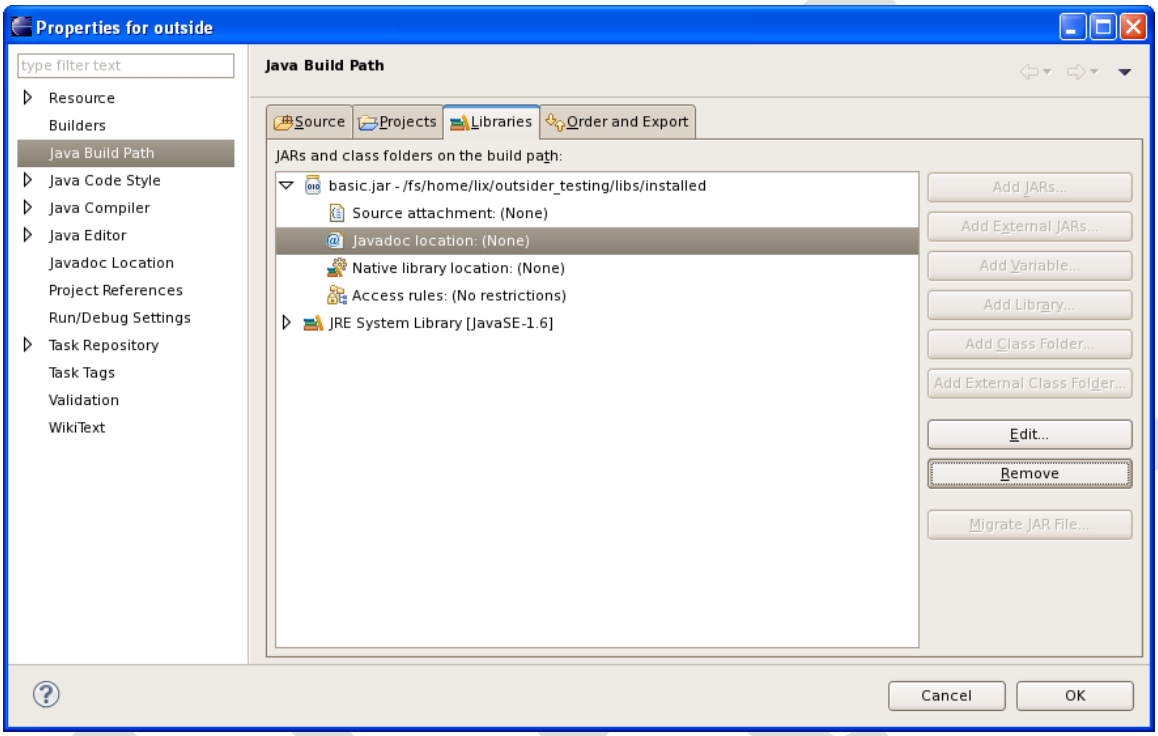
This directory contains the Fortran and C source code for lagk.

The instructions below are based on using Linux and Eclipse Helios version

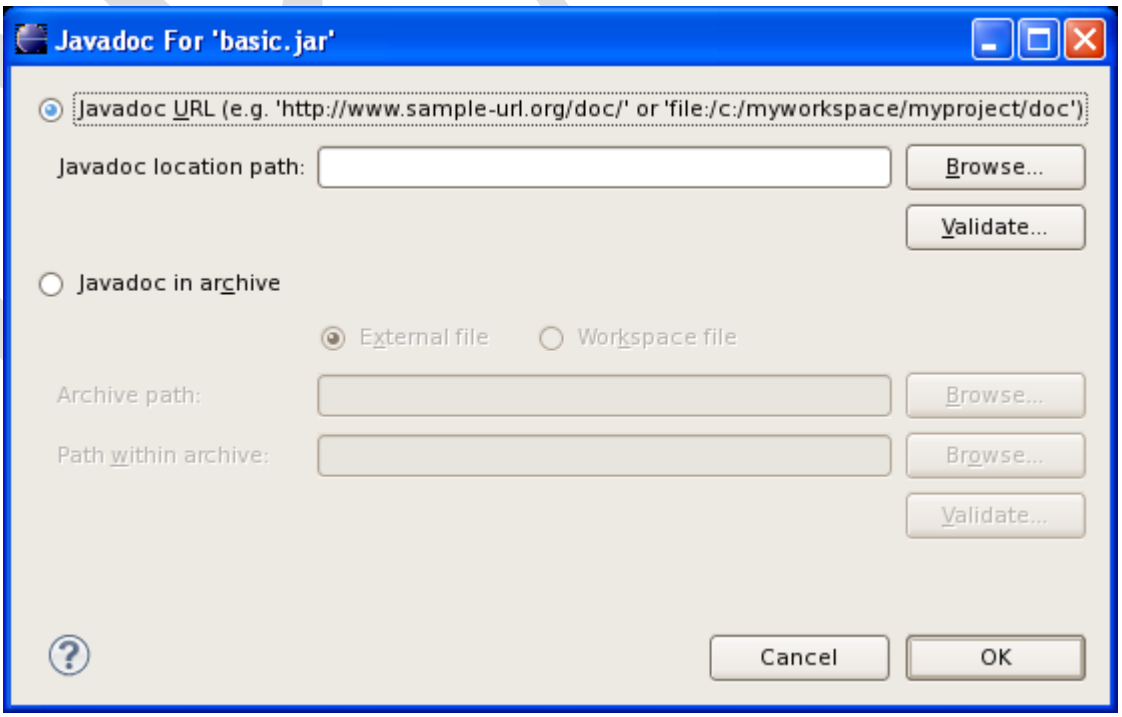
1. Start Eclipse with a new work space. Create a new Java project, e.g. "outside", using the newly generated outside/ directory. Eclipse will show red compiling errors for now.
2. Edit the Eclipse project "outside" properties to add basic.jar:
    a. While highlighting the "outside" project, choose menu bar "Project" → "Properties";
    b. In the pop up window "Properties for outside",
        1. select "Java Build Path" on the left panel,
        2. then choose the tab "Libraries".
        3. Click the button "Add External JARs…"
        4. and add libs/basic.jar.
        5. After clicking OK, All the red errors should go away.



c. It is handy to attach the JAVA API documents to basic.jar so that they can be displayed in Eclipse too:
    1. re-open the window “Properties for outside”,
    2. underneath “basic.jar”, click Javadoc location and
    3. click the button “Edit..”.

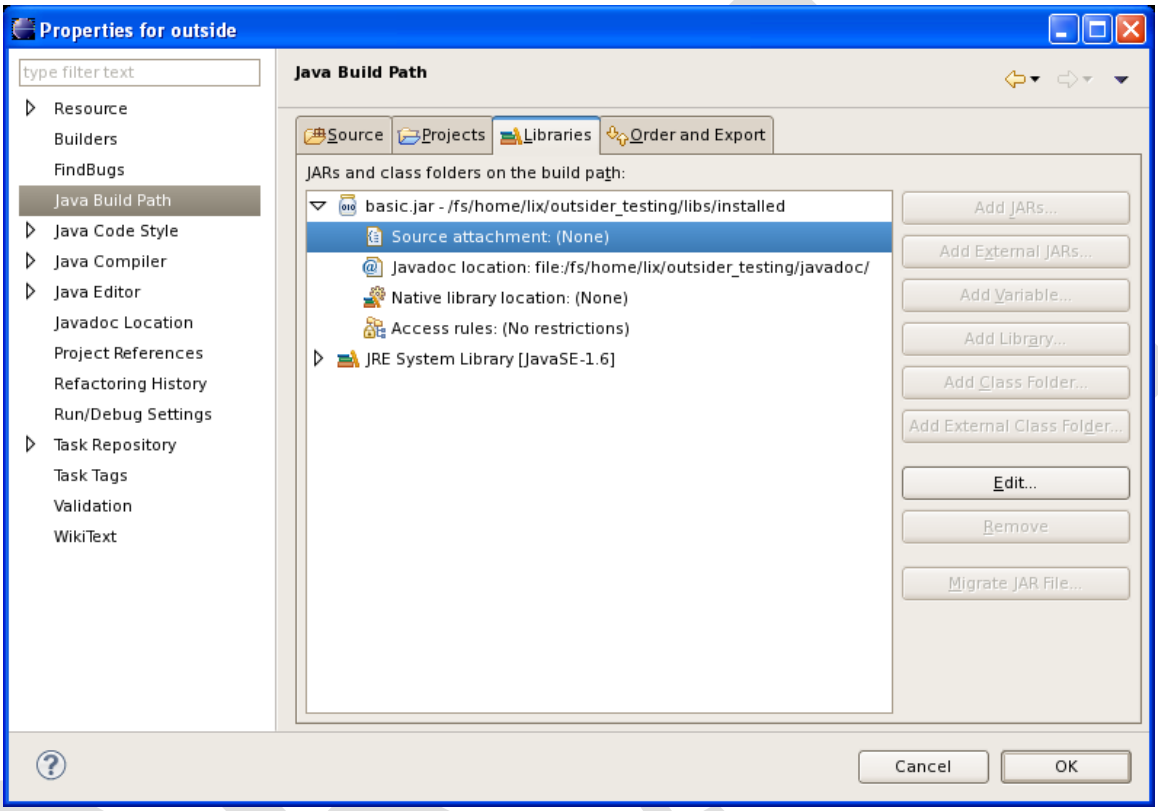


Another pop up window “Javadoc For ‘basic.jar’” shows up:



4. Enter the Javadoc location path to be the javadoc/ directory from the tar file above.
5. Click OK to close the window.
6. Click OK again to close “Properties for outside” window.

d. Similarly, attach the JAVA source code to “basic.jar” file. Highlighting “Source attachment” under “basic.jar” and press the button “Edit..”. In the next popup window, select the directory java_src.



With steps c and d done above, now in Eclipse, when highlighting a class name and press “F3”, the source code will be opened.

Executing the Sample Code and Tests

In the following examples the development and testing frameworks are demonstrated using the OHDFewsadapter and two other made-up "Dummy" adapters.

The OHDFewsadapter is demonstrated using 2 OHD models and 2 non-OHD models. The "Dummy" adapters are demonstrated using made-up "Dummy" models.

1. Tests Using the OHDFEWSAdapter with existing OHD FEWS modules

    - To run OHD models(SacSma or LagK), the second JAR File (sacsma_lagk.jar) is needed.
    - Follow step 3) below to add libs/ sacsma_lagk.jar.
    - No JAVA Api documents for this JAR file.

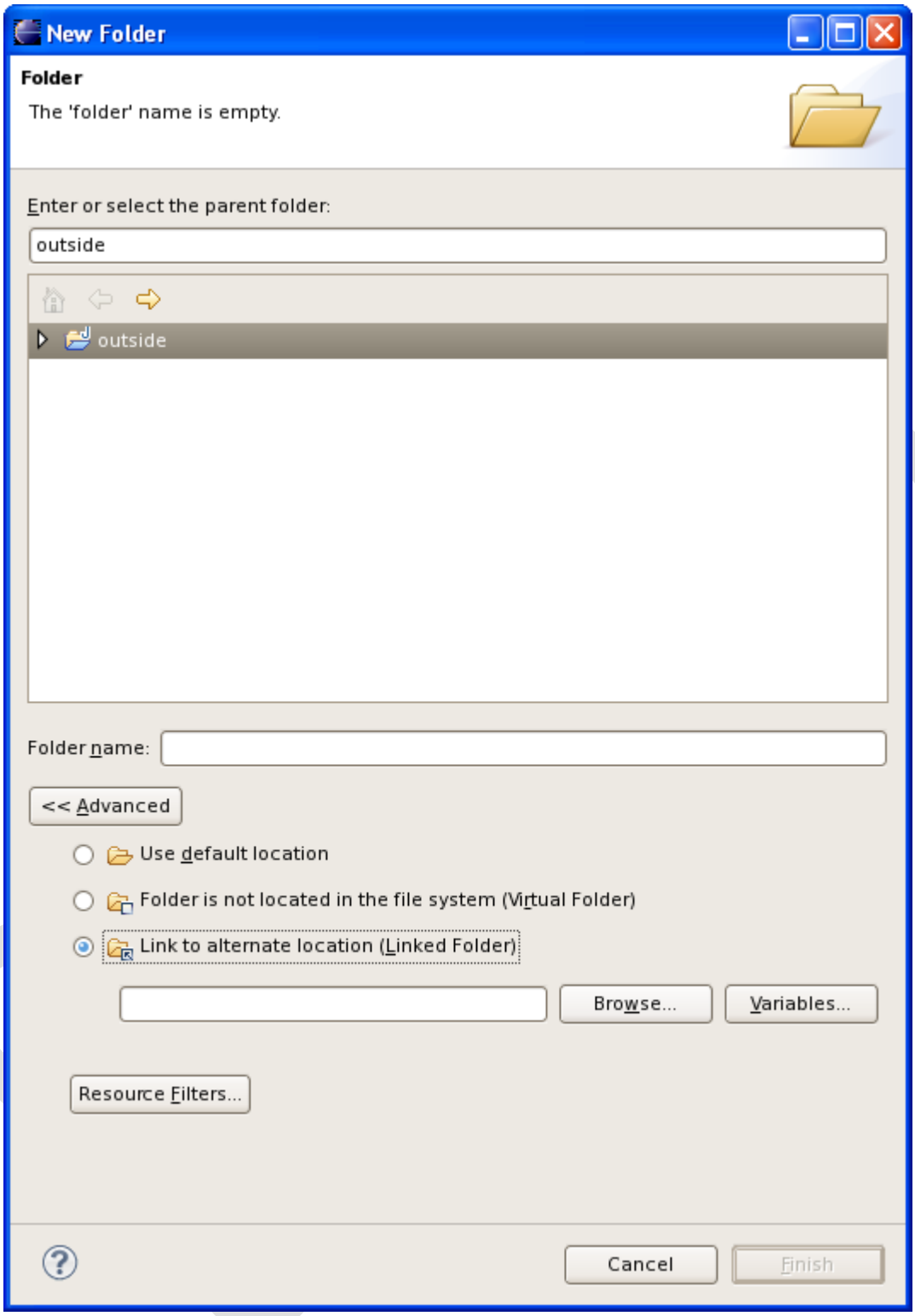
    a. Execute SacSmaTest[1].

    b. Execute LagKTest[2]:
    (This test only works on Linux machine, since the lagk binary executable Modules/bin/lagk can only be run on Linux.)

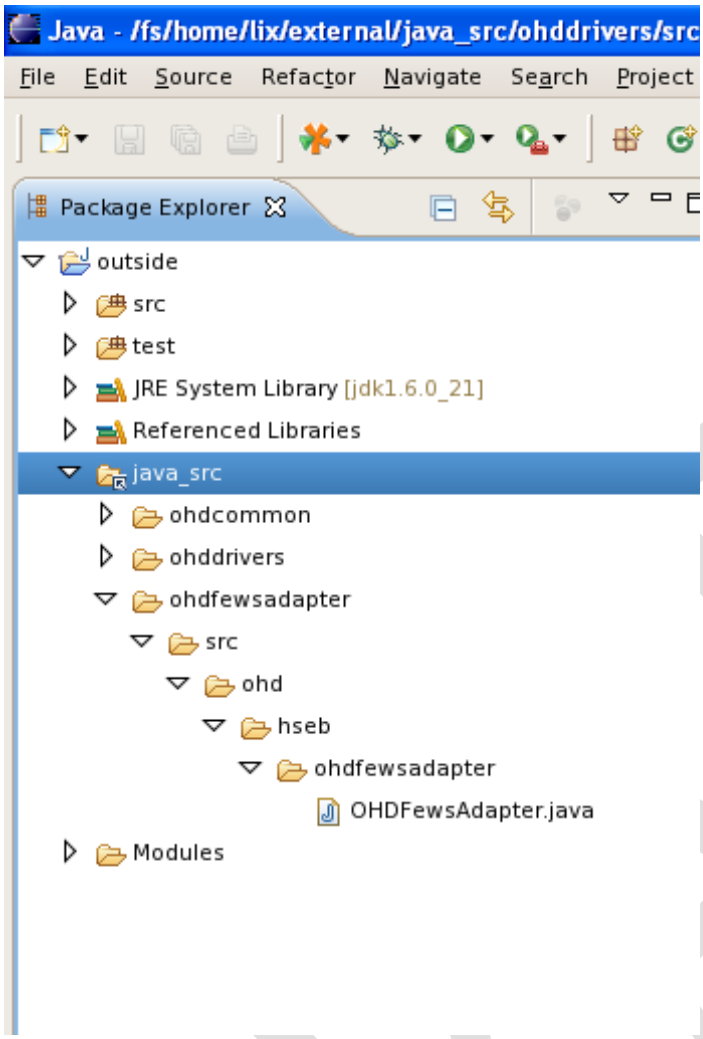
[1] SacSma is written in Java and follows the approach demonstrated in Test #1: public void testDummyModelUsingOHDFEWSAdapter()

[2] LagK is written in C/Fortran and uses the approach demonstrated in Test #2: public void testDummyLegacyModelUsingOHDFEWSAdapter(). All the NWSRFS "Legacy" models ported to FEWS use this approach and use a common framework for interfacing with the Java based LegacyModelAdpater. The framework is made up of a library of C routines. For more information, see the description below.

In order to view the JAVA source code used by the two models, in "Package Explorer", highlighting the project name "outside", right click mouse and select new folder. The following popup window is shown. Expand "Advanced" and select the third radio button "Link to alternate location", then use "Browse.." button to link to java_src directory.



Now, java_src folder was added. Underneath it, there are ohdcommon, ohddrivers and ohdfewsadapter. The class OHDFewsAdapter.java is the entry point for OHD models. Inside ohddrivers, only SacSma and LagK models are included as examples.



2. Tests Using “Dummy” Adapters (two junit tests):

    a. Under src/, please check the JAVA file dummy.DummyAdapter.java and dummy.DummyAdapter2.java. Both represent non-OHD adapters which run their own specific modules.
    b. Under test/, please check the JAVA file dummy.DummyAdapterTest.java containing two junit tests. One for dummy.DummyAdapter.java and one for dummy.DummyAdapter2.java. Only the jar file *basic.jar* is needed.
    c. Run DummyAdapterTest as Junit test and the two tests pass.

Test #1: public void testDummyAdapter()

This test evokes the DummyAdapter which:

a. *prints "Hello World" to the screen. No results verification at the end of the test.*
    - The test shows that OHD testing utilities can be used on any other adapters (e.g. DummyAdapter.java), not only the OHDFewsAdapter.java which OHD modules use.
    - The data files for this test are located at Modules/dummy_module/dummy_basin1.

- Since this dummy module only prints “Hello World”, the output directory “dummy_basin1/output/” is empty. The run info file dummy_basin1/run_info.xml is not used and is a bogus file.

Test #2: public void testDummyAdapter2()

This test evokes DummyAdapter2 which:

a. *copies the file* Modules/dummy_module/dummy_basin2/output_benchmark.xml" to *Modules/dummy_module/dummy_basin2/output/outputs.xml" -- presumed output timeseries file.*
b. *Then the test compares the output time series file "output/outputs.xml" versus the bench mark file "output_benchmark.xml".*
   - This test uses a non-OHD adapter (DummyAdapter2.java) which copies a file to the output directory.
   - Then the test compares the presumed output file against the presumed bench mark file.
   - It shows that the comparison utilities can be used for testing non-OHD modules. The files for this test are located at Modules/dummy_module/dummy_basin2.
   - It uses a text format argument file *args.file*, instead of run_info.xml file which is normally used by the OHD modules in FEWS.

3. Tests using “Dummy” models and OHDFEWSAdapter (two junit tests):

The two tests demonstrate how to use the OHDFewsAdapter to run non-OHD modules (e.g. DummyModelDriver.java and DummyLegacyModelDriver.java). The first model/module is written in Java and the second is a “Legacy” model written using a shell script.

Since OHDFEWSAdapter is controlled by the information inside run_info.xml, it is necessary to introduce this file first. For example, the run info file for running SacSma module on basin amen8 (Modules/sacsma/amen8/run_info.xml) which is really used, contains the following:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <Run xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.wldelft.nl/fews/PI"
xsi:schemaLocation="http://www.wldelft.nl/fews/PI pi_run.xsd" version="1.2">
 3   <timeZone>0.0</timeZone>
 4   <startDateTime date="1988-11-01" time="00:00:00"/>
 5   <endDateTime date="1988-12-01" time="00:00:00"/>
 6   <time0 date="1988-11-22" time="06:00:00"/>
 7   <lastObservationDateTime date="1988-11-22" time="06:00:00"/>
 8   <workDir>Modules/sacsma/amen8/work</workDir>
 9     <inputParameterFile>Modules/sacsma/amen8/input/params.xml</inputParameterFile>
10   <inputStateDescriptionFile>Modules/sacsma/amen8/work/states.xml</inputStateDescriptionFile>
11   <inputTimeSeriesFile>Modules/sacsma/amen8/input/inputs.xml</inputTimeSeriesFile>
12   <outputDiagnosticFile>Modules/sacsma/amen8/output/diag.xml</outputDiagnosticFile>
13   <outputTimeSeriesFile>Modules/sacsma/amen8/output/outputs.xml</outputTimeSeriesFile>
14   <properties>
15     <int key="printDebugInfo" value="0"/>
16     <string key="model" value="ohd.hseb.ohdmodels.sacsma.SacSmaModelDriver"/>
17   </properties>
18 </Run>
```

Line 2: the xml schema.
Line 3: “0.0” is GMT time zone; “-5.0” is East Coast Time.

Line 4: the computation start time
Line 5: the computation end time
Line 6 & 7: the current system time, which is also the last observation time
Line 8: the directory storing temporary files. Note: the final result files are located in output/ directory.
Line 9: the parameter xml file location. In this case, it contains SacSma model parameters for basin AMEN8.
Line 10: the states meta file which contains the exact location of the initial state file and the output state file
Line 11: the input timeseries xml file. In this case, it contains RAIM timeseries.
Line 12 & line 15: line 12 is the diagnostic file. Line 15 controls how much information to be stored in the diagnostic file. If "printDebugInfo" value is "0", only regular information is stored. E.g.:

<line level="3" description="SacSmaModelState validation has passed"/>

level="3" is the regular information level; level="0" is the fatal level, which happens when an Exception occurs during the model run; level="1" is the error level, which is similar to the fatal level; level="2" is the warning level message(the program still runs, but some unusual things happened which needs to bring to the attention of the user.); level="4" is the debug level message, which shows a lot of detailed information during the model run. These message are only logged into the diagnostic file when line 15 "printDebugInfo" value is set to "1".

Line 13: the output timeseries file.
Line 16: this important line directs OHDFewsAdapter to run the specific module. In this case, SacSmaModelDriver.

Test #1: public void testDummyModelUsingOHDFEWSAdapter()

This test is for a module that uses the OHDFewsAdapter. The module extends the OHD Modeldriver class. The module simply takes the input time series and copies it to the output directory. The provided benchmark file (same contents of input file) is then compared to the module generated output file. The time series should match and therefore produce a passing test.

- This test uses the default adapter, OHDFewsAdapter, which calls a non-OHD module, DummyModelDriver.java, determined by the following entry in run_info.xml:

  <string key="model" value="dummy.models.DummyModelDriver"/>

- DummyModelDriver copies a file to the output directory.
- The junit test then compares the generated file versus the bench mark file. The files for this test are located at Modules/dummy_module/dummy_basin3.

Test #2: public void testDummyLegacyModelUsingOHDFEWSAdapter()

This test is for a module that uses the OHDFEWSAdapter. The module extends the OHD LegacyModeldriver class. The legacy model is a simple script takes the input time series and copies it to the output directory. The provided benchmark file (same contents of input file) is then compared to the module generated output file. The time series should match and therefore produce a passing test.

- This test uses the default adapter, OHDFewsAdapter, which calls a non-OHD module(DummyLegacyModelDriver.java) to execute a binary executable (Modules/bin/dummy_executable).
- This dummy binary executable, which is shell script, copies work/ts.txt to work/outpus.txt with slight header change.
- The file outputs.txt is presumed the output timeseries text file generated by real binary executable.

Common Framework Routines

The framework for adding NWSRFS legacy models to FEWS currently exists of several (mostly C and some Fortran) modules intended to be used by all NWSRFS models in FEWS. Figure 1 lists the c modules and the general sequence the stand-alone models will use them in.

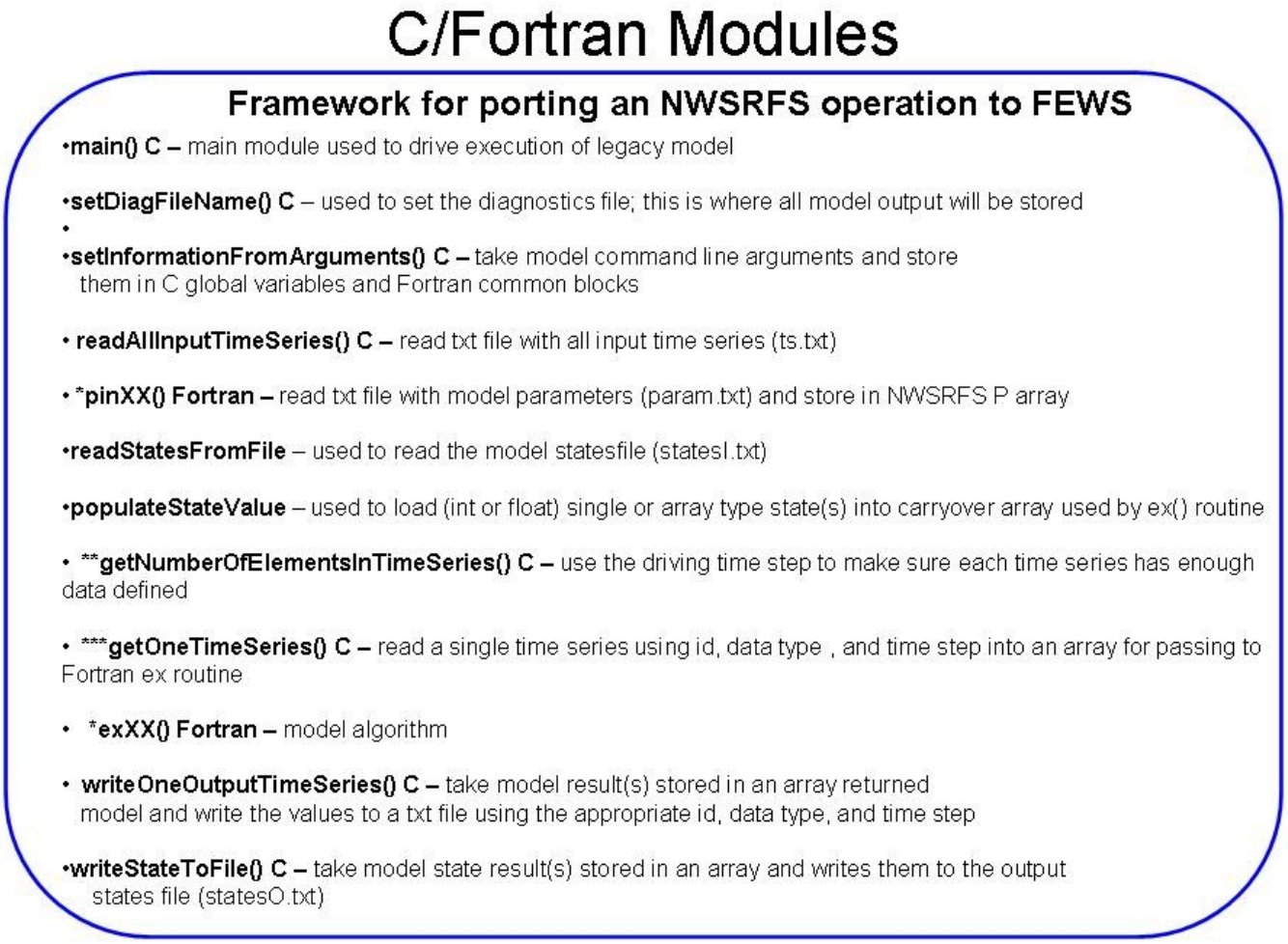


Figure 1: Modules used to add NWSRFS legacy models to FEWS

For the modules listed above the only ones unique to each model are main(), pinXX(), and exXX(). In most cases the pinXX() and exXX() routines can be used as they currently exist. In some cases the pinXX() and exXX() routine need to be preceded by initializing any model specific common blocks.

Routines for Loading/Writing Time series Data

Loading Input Data

An important part of the framework is loading input time series data into data array(s) used by the exXX() routine. Figure 2 shows the overall data flow used to execute an NWSRFS legacy model in FEWS. The right side of the diagram shows the legacy model ingesting a file named “ts.txt”. This file contains all the input data used by the legacy model to execute a run (sample shown in Figure 3 - Note: all times are in Z time zone). On the first line is the total number of time series in the file. Next for each time series has a header containing the identifier (id), data type, time step and number of values in the time series. Finally, the time series values are given in two columns. In the first column is the value in the second column is the date/time. Data is ingested using the module readAllInputTimeSeries(). This routine reads the file “ts.txt” and stores its contents in memory using a C structure. The next step is to read the contents from the structure and store it into data array(s) for passing to the exXX() routine. Reading from the structure is done using the functions getNumberOfElementsInTimeSeries() and getOneTimeSeries().

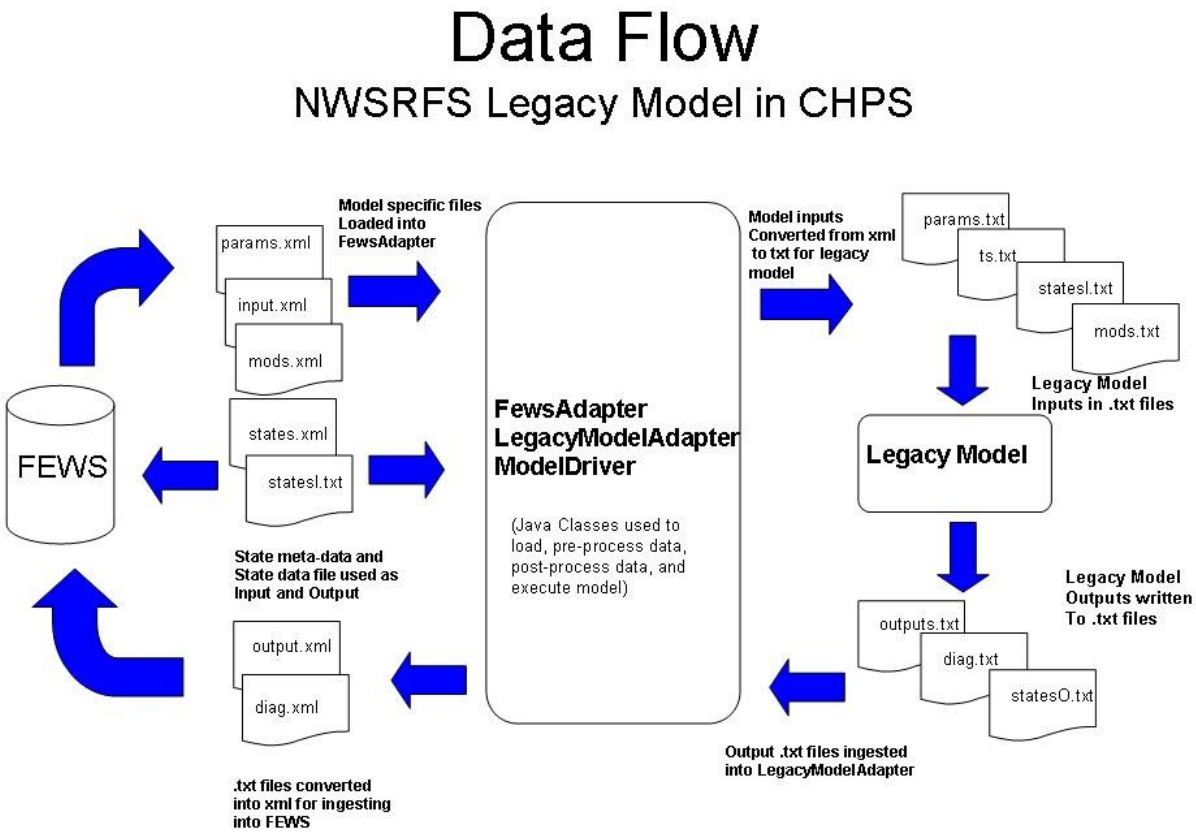


Figure 2: Data Flow Diagram

```
3
SELWEQI1 SQIB  6    4
3588.700000 1993040118
3692.500000 1993040200
3745.000000 1993040206
5974.000000 1993040212
SELWE SELWEQI2  6    4
3692.500000 1993040118
3745.000000 1993040200
5974.000000 1993040206
11718.800000 1993040212
```

SELWE STG 6 4
2.360000 1993040118
2.600000 1993040200
3.600000 1993040206
4.230000 1993040212

Sample ts.txt file for SSARRESV

When called using the driving time series the global variable "outputCount" is defined. This variable is used for writing output time series. The "driving" input time series is considered the driver because its start and end dates coincide with the model run's start and end dates and also the driver's time step can be used to determine how many data values all other input time series should have. In some cases, like SSARRESV, the operation requires all input time series have the same time characteristics (start date, end date, and time step), so any input time series' id, type, and time step can be used for this method. In other cases the operation's documentation specifies a driving time series, so the id, type, and time step for that time series **must** be used in this routine.

Determining Time Series Id, Type, and Time Step

An assumption made by the framework is the time series id, data type and time step used to define time series in the ts.txt file will match the ones defined in the "params.txt" file. An example of a "params.txt" file for the CONSUSE operation is shown below.

PORTNEUF R AT POCATELLO -- DIVERSIONS
0
PIHI1 MAT PIHI1 SQME
PIHI1A SQME PIHI1D SQME
RFIN SQME RFOUT SQME OL SQME CD SQME CE MAPE
42.70 225. 0.60 1.50
0.00 0.00 0.00 0.45 0.60 0.70
0.70 0.65 0.60 0.40 0.00 0.00
0.15 0.003 50.

Sample params.txt file for CONSUSE

The contents of this file define the id, data type and time step for the input and output time series used by the model (for CONSUSE the time step is always 24 hours). The values for each time series are determined after executing the model's pinXX() routine. The job of the pinXX() routine is to parse the file and store model parameters including id, data type , and time step for input and output time series into an internal array (P array). The P array is a Fortran REAL array holding 4 bytes per position. Using the documentation, each developer must determine which slots in the P array contain the id, type, and time step for each input time series prior to calling the methods getNumberOfElementsInTimeSeries() and getOneTimeSeries(). Three utility functions, getTimeSeriesIdFromPArray(), getTimeSeriesCodeFromPArray(), and getIntFromPArray() – located in the common subdirectory, are provided. All of them take the P array and an index as input and return either the id, type or time step.

For example CONUSE has an input MAT time series whose id is stored in slots 20 and 21 (max 8 characters, 4 chars per slot), type (max 4 characters) is stored in slot 22 and the time step is equal to 24.

For some models the number of input and output time series is not consistent for every execution of the model. In these cases the P array location for input and output id, type, and time step is dynamic. See SSARRESV and RES-SNGL for an example of how to determine the id, type, and time step from the P Array.

Writing Output Data

Figure 2 shows the output from the legacy model is written to an output file named “outputs.txt”. This file is similar in structure to the input file shown in Figure 3. It is created by calling the routine writeOneOutputTimeSeries() for each output time series. Similar to the routines used to load input data, this routine requires an id, type, and time step. The values for each output time series should also match the values defined in the “params.txt” file, and therefore, like the input time series id, type, and time step, are determined by querying the P array returned by the pinXX() routine.

For example CONUSE stores the id for the output diversion flow in slots 38 and 39, the data type in slot 40 and the time step is equal to 24.

Routines for Loading/Writing State Data

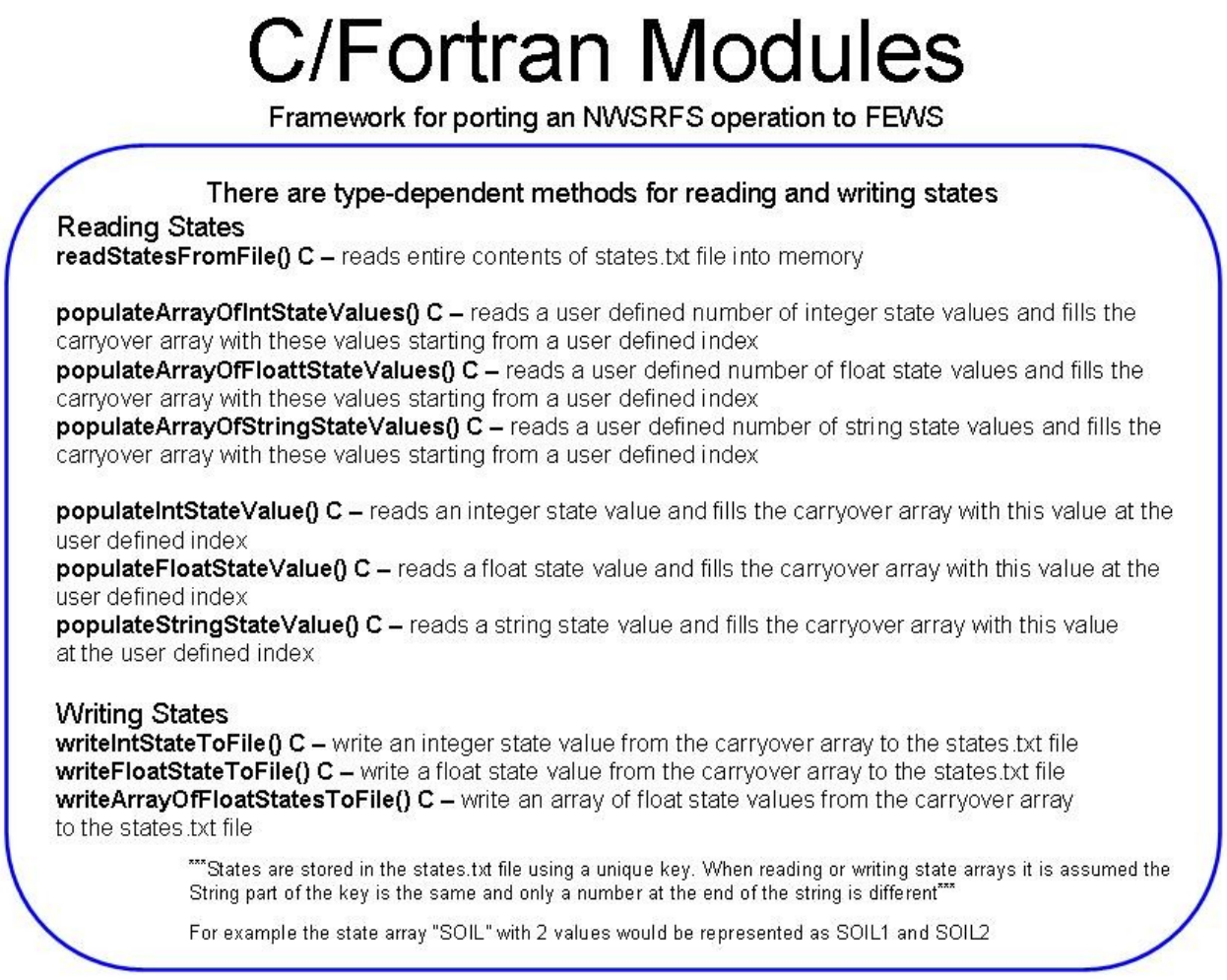


Figure 3: Modules used to read and write states into and out from the legacy models

Loading Input States

The state data used by the stand-alone legacy models are loaded from property type files (i.e. KEY=VALUE). An example is given below. States are loaded prior to entering the model's exXX() routine and are passed into exXX() using a float array (usually named CO).

DOWNSTREAM_RESERVOIR_INST_INFLOW=3588.7
DOWNSTREAM_RESERVOIR_INST_OUTLOW=100.0
DOWNSTREAM_RESERVOIR_ELEVATION=1981.900024
DOWNSTREAM_RESERVOIR_STORAGE=45489.

Sample Contents of State File for SSARRESV Operation

The state values are read using type-dependent C functions shown in Figure 3. The first step is to use the method readStatesFromFile() to load the state file into memory. This method takes as its argument the fully qualified state filename, which is the 3rd command line argument to the model. Next, depending on the dimension of the state (array or scalar value) and type (string, int, or float) the appropriate populateXXX() function is called. Each of these functions takes a key, array to populate, and index as input. In the case of arrays the number of values read is also an input. To read the sample state file shown above four separate calls of the populateFloatStateValue are needed. In each call the same array to be filled is passed in, the only difference is the index of the array the value should be placed in. Developers should reference the operation's section VIII documentation to know the index for each state value.

When loading state arrays, the starting location for loading values into the array and the number of values to load is passed into the populateArrayXXX() function. Note: array type states are stored in the state file as separate KEY=VALUE entries. A unique number at the end of the key is used to differentiate the states. For example a state array representing the soil temperature (using key SOILT) would have 2 entries (SOILT1 and SOILT2) if two soil temperatures are needed.

Writing Output States

Similarly, states are written to a text file after returning from the exXX() routine using the type dependent functions shown in Figure 3. In most/all cases the type of write function used will depend on the type of read function used. For example, to write the statues at the end of the run for the states shown in Figure 3a, four separate calls to writeFloatStateToFile() is used. When righting a single float value, the float value to write is passed in (e.g. CO(4)). On the other hand, when writing a state float array, the number of values to read and the CO array and indexed to the position where values should be read from. It is then passed to writeArrayOfFloatsToFile(). For example, the Tatum model writes states using the following code.

```
writeArrayOfFloatStatesToFile(outputStateFilePtr, "CMS", carray, 1, numberLayers-1);
```

In this case numberLayers-1 float values are written to the file using the contents of the 'carray' starting at positon '1'.

Routines for Writing Diagnostics

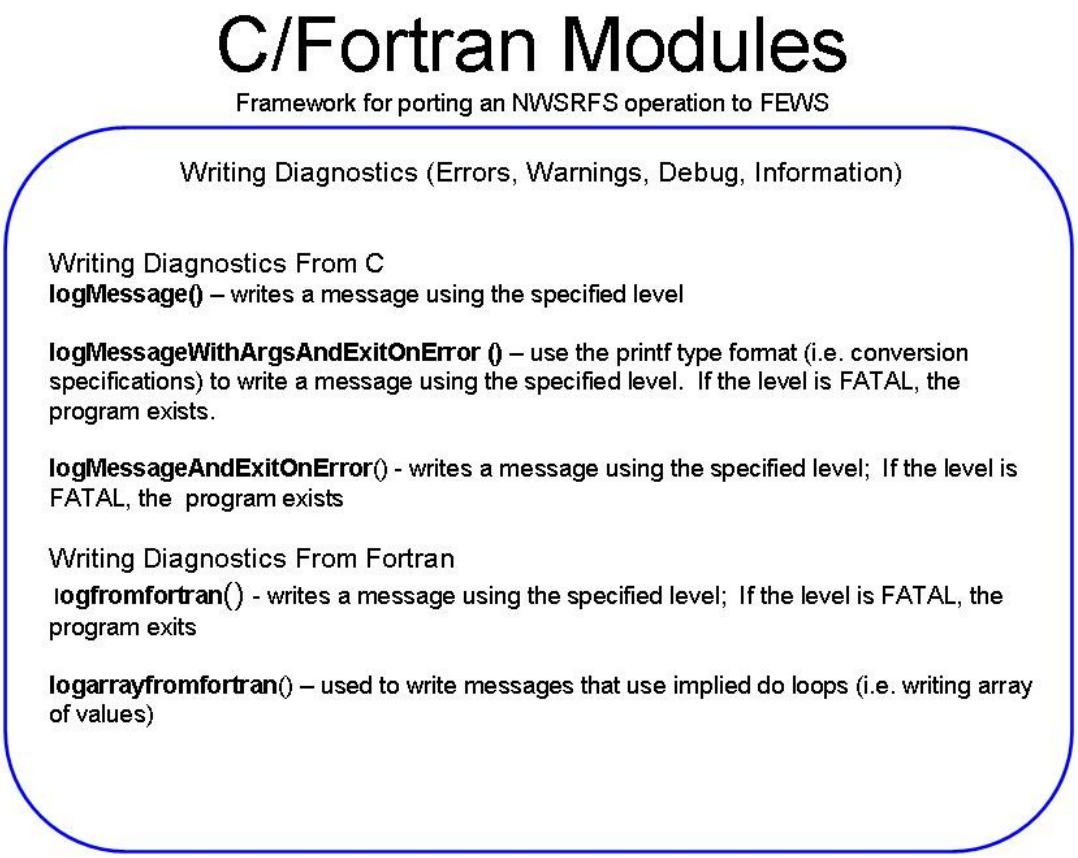


Figure 3A: Diagnostic Functions

The C and Fortran diagnostic functions shown in Figure 3A are used to write error, warning, debug, and normal output. Similar to output time series, the approach taken for diagnostics is for the legacy models to populate a text file (diag.txt) with diagnostics and then use the OhdFewsadapter to convert diag.txt file to a diag.xml used by FEWS.

Most of the diagnostics written will be from existing warning, debug, and error messages the legacy models already have. The remaining diagnostics are produced by the C and Fortran routines in this framework. The following diagnostic levels are available:

- DEBUG_LEVEL
- INFO_LEVEL
- WARNING_LEVEL
- FATAL_LEVEL

In C there are separate functions for forcing the program to exit after logging. In Fortran the program will exit when logging using FATAL_LEVEL. The approach for retrofitting legacy model diagnostics is described below.